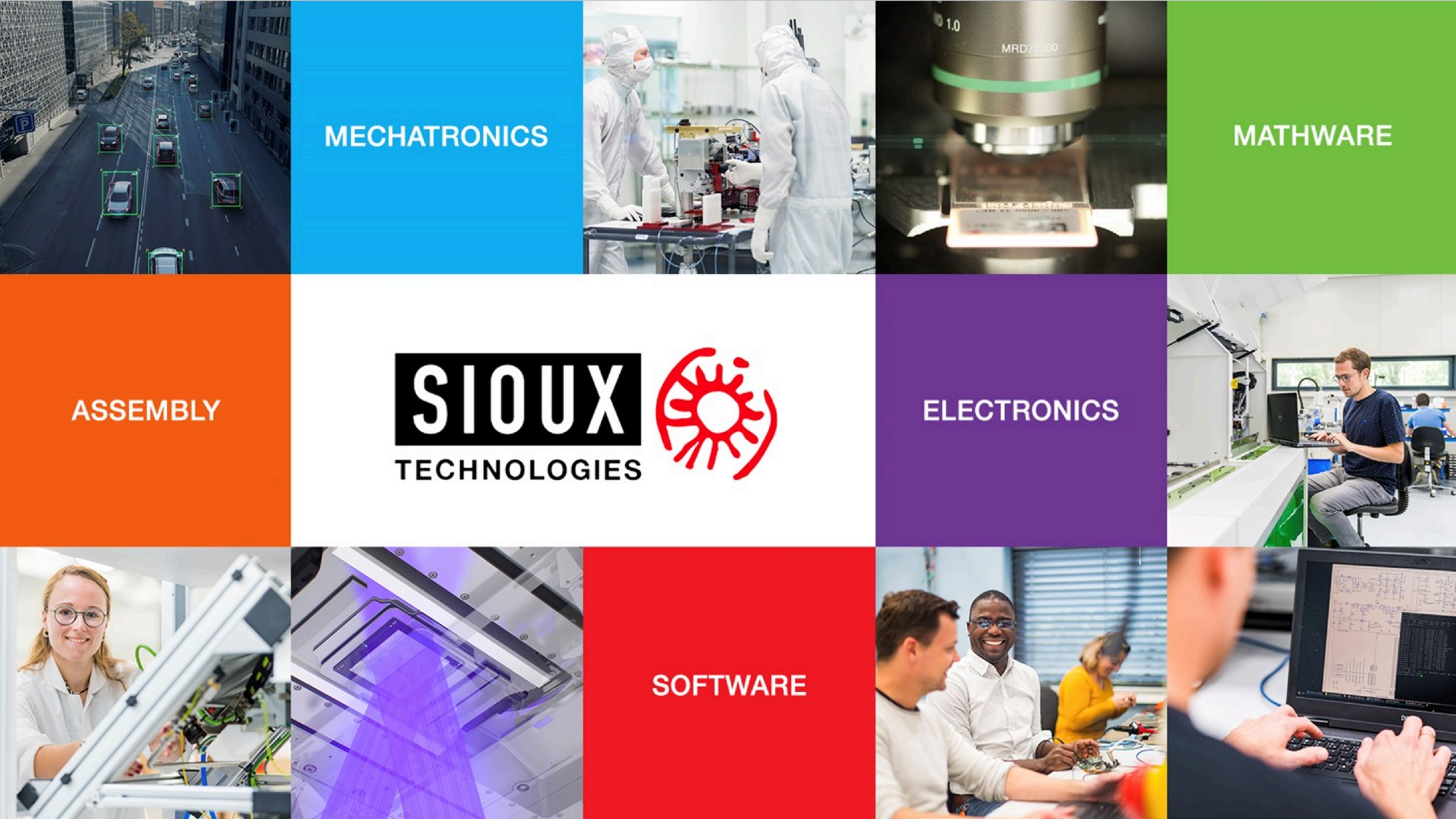
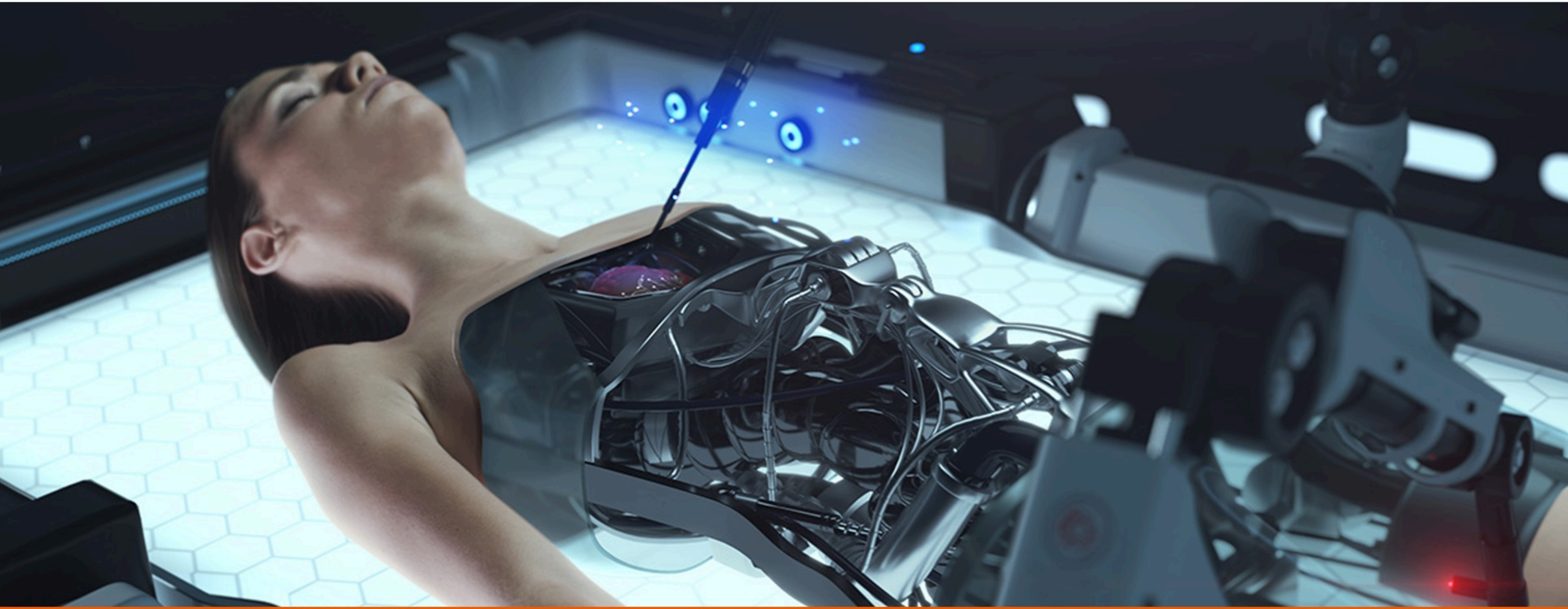

MECHATRONICS
MATHWARE
ASSEMBLY
SIOUX
TECHNOLOGIES
ELECTRONICS
SOFTWARE

C++ Training

Compile-time polymorphism



What is Compile-Time Polymorphism?

- Compile-time type erasure
- Avoid code duplication
- Meta-programming; type inspection
- **Templates**

Key features

- C++99/03
  - Class/Function templates
  - Template specialization
- C++11/14/17 era
  - Variable templates
  - Auto keyword
  - constexpr
  - SFINAE
- C++20
  - Concepts

Function template

```cpp
template <typename T>
T add(T const& a, T const& b)
{
    return a + b;
}
int main()
{
    int result = add(5, 10);   // 5 is int
    double res = add(5.5, 2.3); // 5.5 is double
}
```

Does not promote or demote the built-in type. No casting

Function template

```
template <typename T>
T add(T const& a, T const& b)
{
    return a + b;
}
int main()
{
    /*???*/ res = add(5, 2.3);
}
```

Question : What will type T be deduced to?

Class template

```
template <typename T, std::size_t SIZE>
struct Array
{
    T& operator[](std::size_t index)
    {
        return data[index];
    }
private:
    T data[SIZE];
};

int main()
{
    Array<int, 10> intArray;
    Array<double, 5> dblArray;
}
```

Size of the array is part of type, std::array in c++11

Partial class template specialization

```cpp
template <std::size_t SIZE>
struct Array<BigType, SIZE>
{
    BigType& operator[](std::size_t index)
    {
        return data[index];
    }
private:
    Store<SIZE> data;
};
```

Function template overloading

```cpp
template <typename T, typename S>
void print(T const& value, S& stream)
{
    stream << "Value: " << value << std::endl;
}
template <typename S>
void print(std::string const& value, S& stream)
{
    stream << "std::string specialization: " << value << std::endl;
}
int main()
{
    print(123, std::cout);
    print("combust", std::cout);
}
```

Question: What is the output of both print statements?

Full Function template specialization

```
template <typename T, typename S>
void print(T const& value, S& stream)
{
    stream << "Value: " << value << std::endl;
}
template <>
void print<std::string, std::ostream>(std::string const& value, std::ostream& stream)
{
    stream << "std::string specialization: " << value << std::endl;
}
int main()
{
    print(123, std::cout);
    print("combust", std::cout);
}
```

Partial template specialization for functions is not possible in C++.

Generally, overload iso specialize functions.

Primary function template with partial class specialization

```cpp
template<typename T, typename U>
class Operation {
public:
    static void perform(T value, U extra) {
        std::cout << "Generic operation with type T and U: " << value << ", " << extra << '\n';
    }
};

// Partial specialization for when the first parameter is a pointer
template<typename U>
class Operation<int*, U> {
public:
    static void perform(int* value, U extra) {
        std::cout << "Operation for pointer to int. Value: " << *value << ", Extra: " << extra << '\n';
    }
};

// Function template that dispatches to the Operation class
template<typename T, typename U>
void executeOperation(T value, U extra) {
    Operation<T, U>::perform(value, extra);
}
```

C++20 - auto keyword 1/2

```
auto /*C++14 auto*/ add(auto const& a, auto const& b /*C++20 auto*/)
{
    return a + b;
}
auto add(std::string const& a, std::string const& b)
{
    return std::stoi(a) + std::stoi(b);
}
int main()
{
    auto result = add(5, 10); //C++11 auto
    auto res = add("1", "2");
}
```

Same issue as previous slide

C++20 - auto keyword 2/2

```cpp
#include <iostream>
auto add(auto const& a, auto const& b)
{
    return a + b;
}
int main()
{
    auto result = add(6.6, 10);
    std::cout << typeid(result).name() << std::endl;
}
```

What is the type of 'result'? types of a and b are deduced independently.

Variable templates - C++14

```
template<class T>
constexpr T pi = T(3.1415926535897932385L);
template<class T>
T circularArea(T r) {
    return pi<T> * r * r;
}
```

vs

```
template <typename T>
struct PI {
    PI():value(3.1415926535897932385L){}
    static const T value;
}
PI<T>::value
```

exercises

exercises/compile_time_polymorphism/ex1.cpp

Exercise on template specialization and overloading

SFINAE

Substitution Failure Is Not An Error

1. Create a set of all possible candidates including non-specialized
2. Substitute type
3. No compilation error, if substitution fails
4. Remove entry from candidate set, if substitution fails

Candidate set

```
template <typename T, typename Stream>
auto print(T const& value, Stream& stream)
{
    stream << "std::string specialization: " << value << std::endl;
    return std::begin(value);
}
template <typename Stream>
void print(double value, Stream& stream)
{
    stream << "Value: " << value << std::endl;
}
int main() {
    print(123, std::cout);
    print(std::string("combust"), std::cout);
}
```

Overload resolution set

Remove primary candiate from set. This compiles.

```
template <typename T, typename Stream>
auto print(T const& value, Stream& stream) -> T::iterator
{
    stream << "std::string specialization: " << value << std::endl;
    return std::begin(value);
}
template <typename Stream>
void print(double value, Stream& stream)
{
    stream << "Value: " << value << std::endl;
}
```

Overload resolution set

- SFINAE only works in the 'immediate' context.
- Generally, think arguments and return type.
- Body of a function is NOT the immediate context.

Fails to compile - no SFINAE:

```
template <typename T, typename Stream>
auto print(T const& value, Stream& stream)
{
    typename T::iterator begin; // NOT the immediate context
    stream << "std::string specialization: " << value << std::endl;
    return std::begin(value);
}
```

std::enable_if - C++11

```cpp
namespace std
{
    //Primary template for enable_if, does NOT wrap type T
    template<bool B, class T = void>
    struct enable_if {};
    //Specialization for B==True, does wrap type T.
    template<class T>
    struct enable_if<true, T> { typedef T type; };
}
```

std::enable_if example

```
template <typename T, typename Enable = typename std::enable_if_t<!std::is_integral<T>::value>>
void print(T const& value)
{
    std::cout << "Primary template: " << value << std::endl;
}
template <typename T>
typename std::enable_if_t<std::is_integral<T>::value> print(T const& value) {
    std::cout << "Integral value: " << value << std::endl;
}
```

::value is true for std::is_integeral struct if T is integeral.

::value static const member -> value is known at compile time.

Compiler errors

Assume there is no primary candidate implementation

```
<source>:11:10: error: no matching function for call to 'print(double)'
   11 |     print(42.0);      // Works for integers
      |     ~~~~~^~~~~~
<source>:6:59: note: candidate: 'template<class T> typename std::enable_if<std::is_integral<_Tp>::value>::type print(const T&)'
    6 | typename std::enable_if<std::is_integral<T>::value>::type print(const T& value) {
      |                                                           ^~~~~
<source>:6:59: note:   template argument deduction/substitution failed:
<source>: In substitution of 'template<class T> typename std::enable_if<std::is_integral<_Tp>::value>::type print(const T&) [with T = double]':
<source>:11:10:   required from here
   11 |     print(42.0);      // Works for integers
      |     ~~~~~^~~~~~
<source>:6:59: error: no type named 'type' in 'struct std::enable_if<false, void>'
    6 | typename std::enable_if<std::is_integral<T>::value>::type print(const T& value) {
      |
```

Custom compiler errors

Assume multiple print implementations

```
template <typename T, typename = typename std::enable_if<!std::is_integral<T>::value>::type>
void print(T const& value)
{
    static_assert(false, "Unsupported print operation");
}

template <typename T>
typename std::enable_if_t<std::is_integral<T>::value> print(T const& value)
{
    std::cout << "Integral value: " << value << std::endl;
}
```

std::void_t - C++17

```
template< class... >
using void_t = void;
```

- will reduce to void if substitution succeeds
- will remove candidate from set if substitution fails.

std::void_t - example

```
template <typename, typename = std::void_t<>>
struct has_type_member : std::false_type {};
template <typename T>
struct has_type_member<T, std::void_t<typename T::type>> : std::true_type {};
struct WithType {
    using type = int;
};
struct WithoutType {};

template <typename T>
constexpr bool has_type_member_v = has_type_member<T>::value;
int main() {
    std::cout << std::boolalpha << "WithType has type member: " << has_type_member_v<WithType> << '\n';
    std::cout << "WithoutType has type member: " << has_type_member_v<WithoutType> << '\n';
}
---
WithType has type member: true
WithoutType has type member: false
```

std::void_t expected output

```
#include <iostream>
#include <type_traits>
template <typename, typename = int>
struct has_type_member : std::false_type {};
template <typename T>
struct has_type_member<T, std::void_t<typename T::type>> : std::true_type {};
struct WithType {
    using type = int;
};
struct WithoutType {};
template <typename T>
constexpr bool has_type_member_v = has_type_member<T>::value;
int main() {
    std::cout << std::boolalpha << "WithType has type member: " << has_type_member_v<WithType> << '\n';
    std::cout << "WithoutType has type member: " << has_type_member_v<WithoutType> << '\n';
}
---
WithType has type member: false
WithoutType has type member: false
```

std:declval - C++11

```
template <typename T>
auto call_example() -> decltype(std::declval<T>().example(), void()) {
    std::cout << "T has example() method\n";
}
```

Instantiate type, even if no constructor available

comma operator do not use result of .example()

decltype(...expression...)

constexpr - C++17

```cpp
template <typename T>
void print(T const& value)
{
    if constexpr (std::is_integral_v<T>)
    {
        std::cout << "Integral value: " << value << std::endl;
    }
    else
    {
        std::cout << "Non integral value: " << value << std::endl;
    }
}
```

constexpr - decltype

But what if you are using 'auto' and do not know type 'T'?

```
void print(auto const& value)
{
    if constexpr (std::is_integral_v<decltype(value)>)
    {
        std::cout << "Integral value: " << value << std::endl;
    }
    else
    {
        std::cout << "Non integral value: " << value << std::endl;
    }
}
```

constexpr

```
template <typename T>
void print(T const& value)
{
    if constexpr (value == 124)
    {
        std::cout << "Integral value: " << value << std::endl;
    }
    else
    {
        static_assert(false, "Unsupported print operation");
    }
}
```

```
<source>:5:5: error: 'value' is not a constant expression
    5 |     if constexpr (value == 124)
      |     ^
```

exercises

exercises/compile_time_polymorphism/ex2.cpp

Concepts - C++20

```
#include <concepts>
template <typename T>
concept Integral = std::is_integral_v<T>;
void print(auto const& value)
{
    static_assert(std::is_integral_v<decltype(value)>, "Unsupported print operation");
}
void print(Integral auto const& value)
{
    std::cout << "Integral value: " << value << std::endl;
}
```

Concepts

Compiler errors

Assume there is no primary candidate implementation

```
<source>: In function 'int main()':
<source>:13:10: error: no matching function for call to 'print(double)'
   13 |     print(42.0);      // Works for integers
      |     ~~~~~^~~~~~
<source>:7:6: note: candidate: 'template<class auto:1>  requires  Integral<auto:1> void print(auto:1)'
    7 | void print(Integral auto value)
      |      ^~~~~
<source>:7:6: note:   template argument deduction/substitution failed:
<source>:7:6: note: constraints not satisfied
<source>: In substitution of 'template<class auto:1>  requires  Integral<auto:1> void print(auto:1) [with auto:1 = double]':
<source>:13:10:   required from here
   13 |     print(42.0);      // Works for integers
      |     ~~~~~^~~~~~
<source>:5:9:   required for the satisfaction of 'Integral<auto:1>' [with auto:1 = double]
<source>:5:25: note: the expression 'is_integral_v<T> [with T = double]' evaluated to 'false'
    5 | concept Integral = std::is_integral_v<T>;
      |
```

Custom concepts

```
template <typename T>
void print(T const& value) requires requires(T a) {  std::cout << a }
{
    std::cout << "Value: " << value << std::endl;
}
```

requires clause + requires expression

also works with auto

compile-time!

Combination of requires and predicate

```
template <typename T>
void calculate_sum(T const& a, T const& b) requires (std::is_arithmetic_v<T> || requires(T x, T y) {
    x + y;
})
{
    std::cout << "Sum: " << a + b << std::endl;
}
```

Constrained class template

```
template <typename T>
concept Addable = requires(T a, T b)
{
    { a + b } -> std::same_as<T>;
};
template <Addable T>
struct Calculator
{
    Calculator(T value) : m_value(value) {}
private:
    T m_value;
};
```

exercises

exercises/compile_time_polymorphism/ex3.cpp

We bring **high-tech** to life